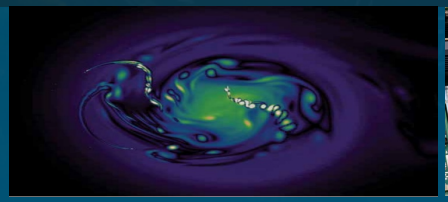
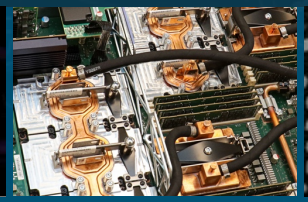
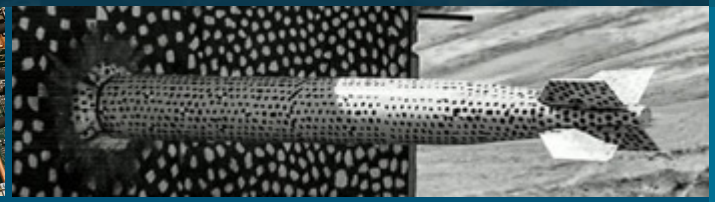
# Analyzing Build System Pressure for the ASC Program








*PRESENTED BY*

Simon D. Hammond (sdhammo@sandia.gov)



UNCLASSIFIED UNLIMITED RELEASE - SAND2018-10935 PE

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.



Images courtesy of Sandia National Laboratories and Oak Ridge National Laboratory

# Introduction

**The L2 Codesign Milestone for 2018 is about compilers and build systems**

Lots of anecdotal evidence from Trilinos and ATDM application users that build times are becoming a significant problem for productivity

- Major issue is that we do not have effective data on where time in builds go
- Need to understand what we are building, how and where novel technologies (from the lab and vendors can help = codesign)

**This talk covers three areas:**

- Basic compile and link times (understand where compile time goes)
- Memory required for compilation/build (help design higher productivity user nodes)
- Object sizes, particularly for debug (help design better support tools)

# Observations from the Last Five Years

**Codes continue to grow larger and more complex each year**

- Do not typically see pruning of physics modules and features
- Creates a lasting burden on the ASC program to maintain complex code bases into the future – expensive and man-power heavy

Environments for applications are changing rapidly

- Trinity KNL Platform was a significant increase in level of development effort over previous generations of machines (more threading and vectorization, not easy programming model match)
- LLNL/Sierra is likely to be another significant step function in code design and implementation – GPU kernels, more careful data structure design. Perhaps slightly clearer programming model choices? OpenMP, Kokkos, RAJA, FleCSI, CHAI etc.
- Future set to be equally (more?) complicated for development

# Observations from (Sandia's) Code Teams

**Significant growth in the complexity of C++ features being used**

- Partially driven by adoption of Kokkos, RAJA and FleCSI (but also better STL etc)
- Greater compiler maturity allowing standards to be adopted
- Stronger push by code teams to have later standards be an option

Much greater use of C++ templating, Kokkos/RAJA/FleCSI and Header Files

- Frameworks are appearing in much broader range of application codes
- Seeing C++ templates being used for type definitions to allow for complex types (wide doubles, SIMD types and Sacado FADs/UQ)
- Pushes more kernels to be made available via header files or use of explicit template instantiation
- Profound change in compiler assumptions

# Effects on Code Development

Anecdotal evidence of significant increases in compile times

- Even for small to moderate changes in some functions
- Templating/ETI leading to longer compile times as functions re-generated multiple times for each type
- History of various packages needing specific types means we compile multiple variants

**Effect on developer productivity can be dramatic**

- Long times to check if code will compile and run successfully (now required prior to commits for some code packages)
- Longer automatic pull-request integration times for Git-based application projects
- Harder and more complex to profile and debug

# Principal Questions for the L2

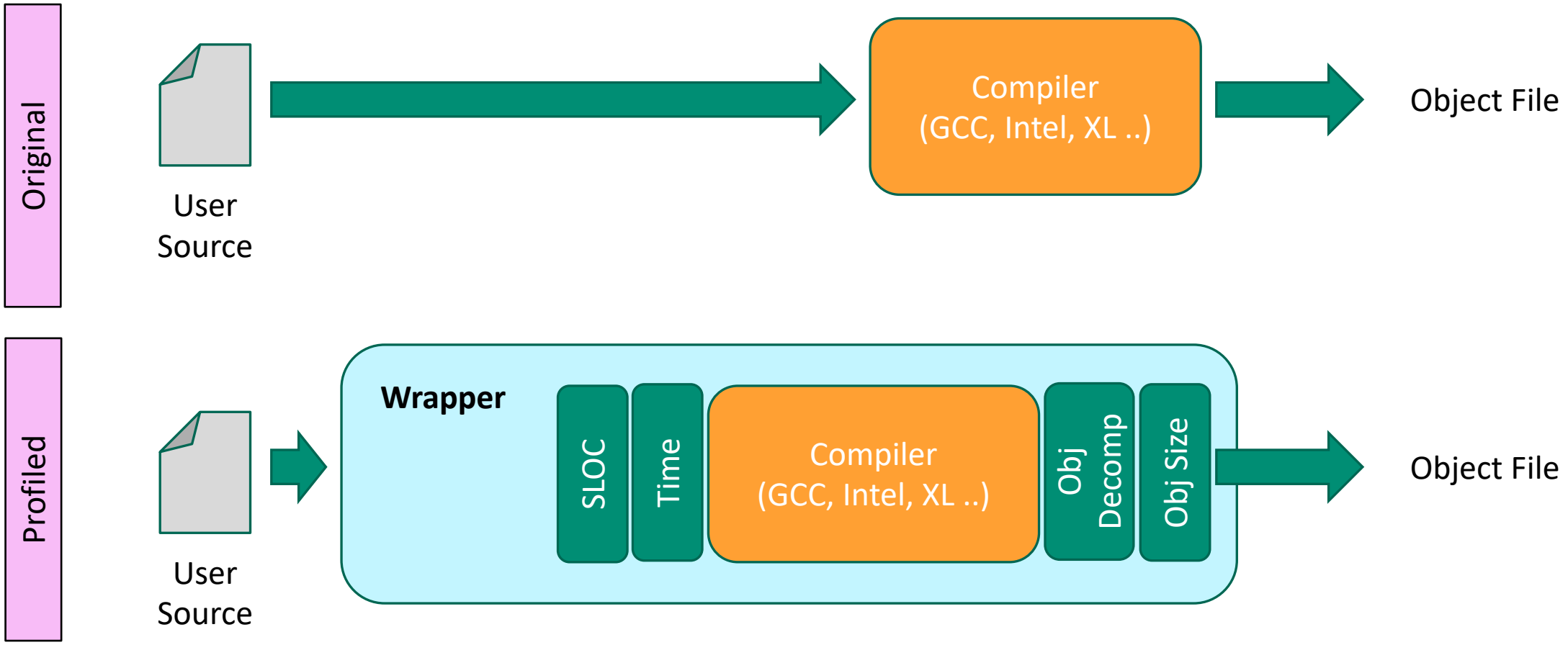
**What does a build for a modern ASC application/code base look like?**

- Compile times
- Object sizes
- Lines of code
- Memory requirements

**What if anything can we recommend for workstations/user service nodes to address these concerns?**

# Build System Profiling

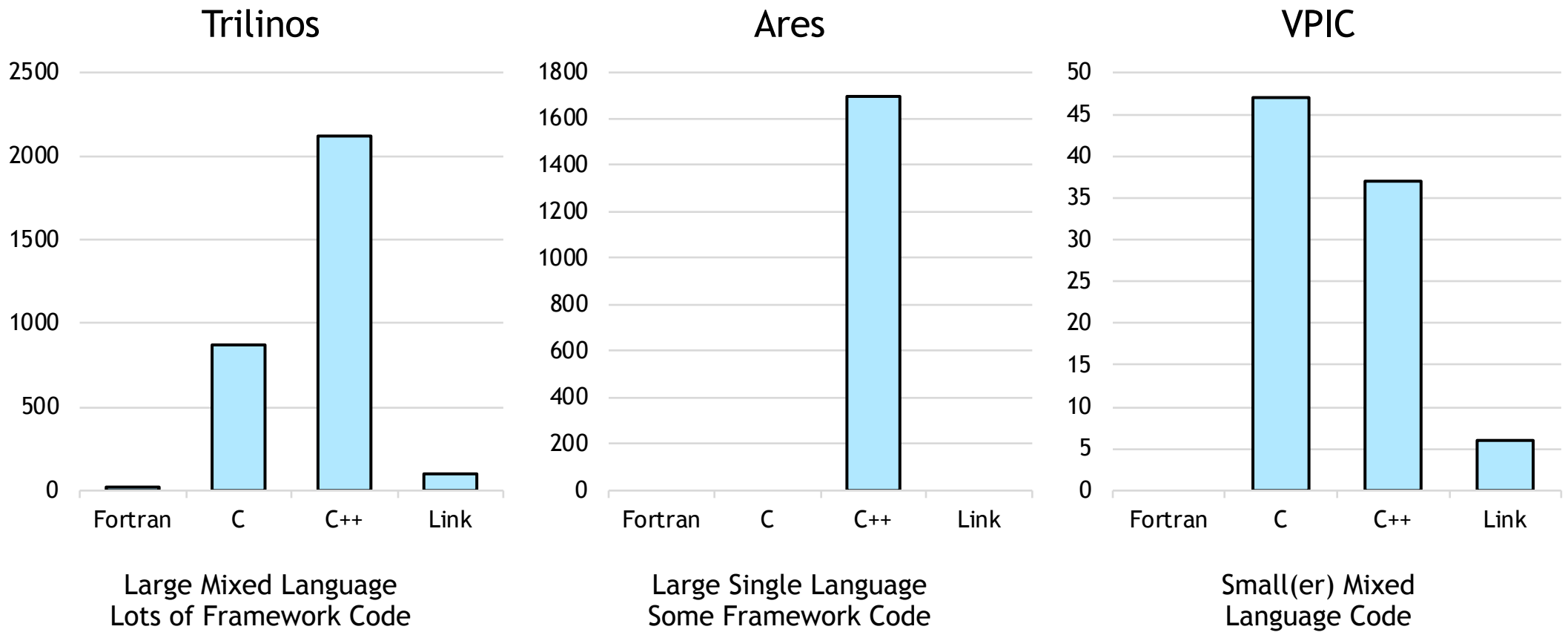
# Build System Profiling




Can wrap standard compilers (e.g. GCC, Intel, etc) or wrap from within MPI compiler chain (mpicc, mpicxx, mpif90..)

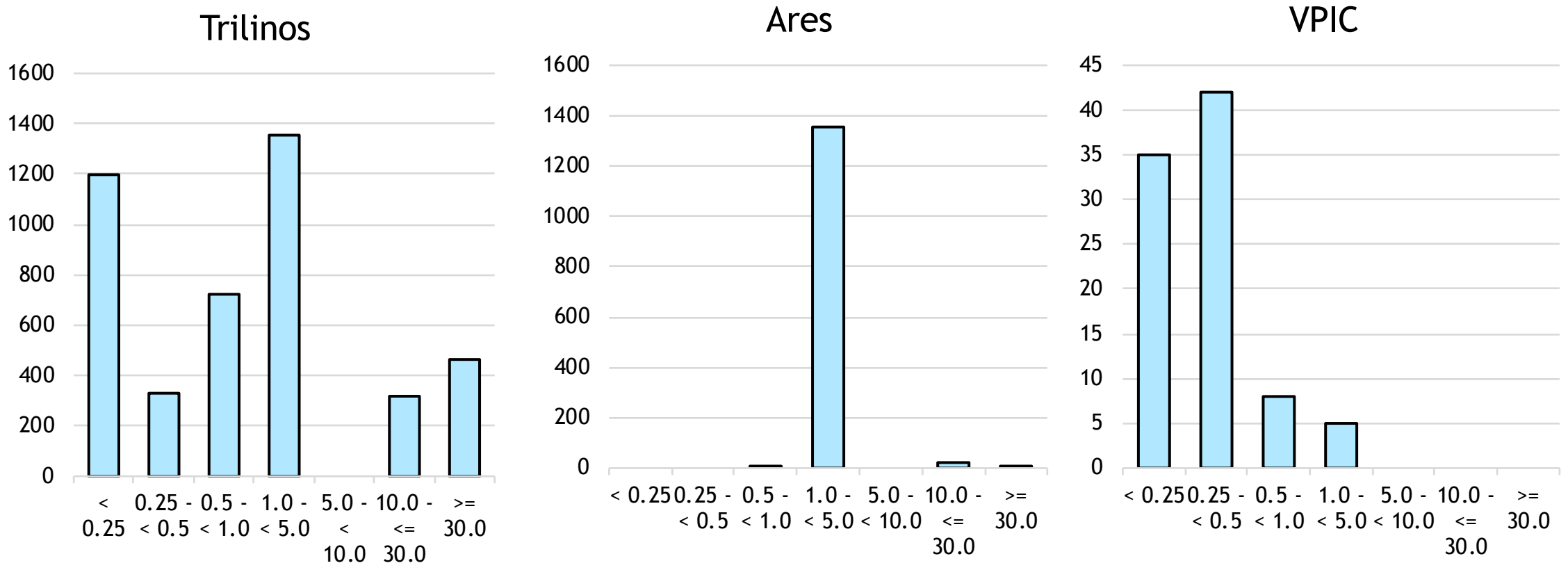
# Results and Analysis

# Multi-Lab Code Comparison




Use variety of example codes from across the trilabs (Trilinos (Solvers) from Sandia, Ares (Hydrocode) from LLNL and VPIC (Particle-in-Cell) from LANL

# Code Compile and Link Times




Breakdown by compile and link time in seconds

Significant variation in compile times across code bases, max times:
Trilinos = 907.85, Ares = 157.55, VPIC = 2.27

# User Service/Compile Nodes – Memory Requirements

# User Service Node Memory Requirements

**Majority of users want to be able to compile code quickly and efficiently on platforms**

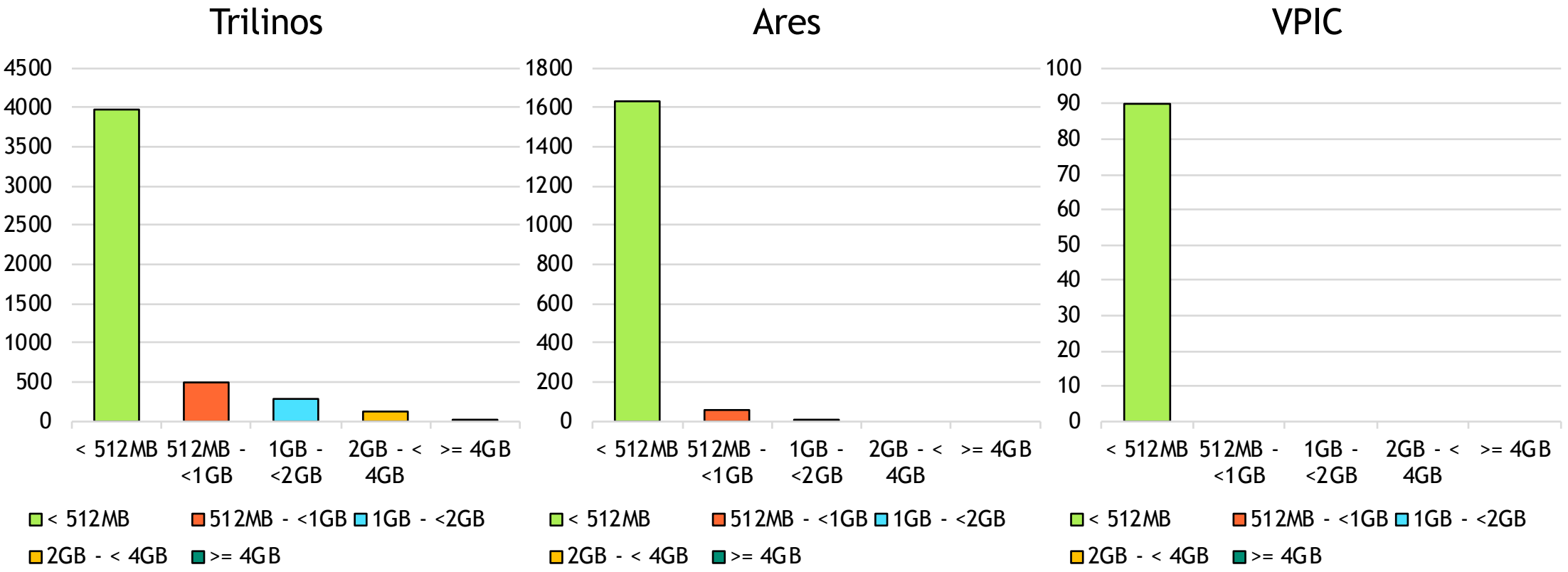
- Typical for users but also automated continuous integration and overnight testing
- Anecdotal reports of compilers crashing (segmentation-faults) when compiler large projects
- Causes have traced to memory exhaustion

Effects are that larger code projects can take longer to compile when restricting number of codes on service nodes

- More memory would allow us to run more parallel builds
- Effects uncertain when more complex requirements are encountered

**How much memory do large projects requirement for compilation on user service nodes?**

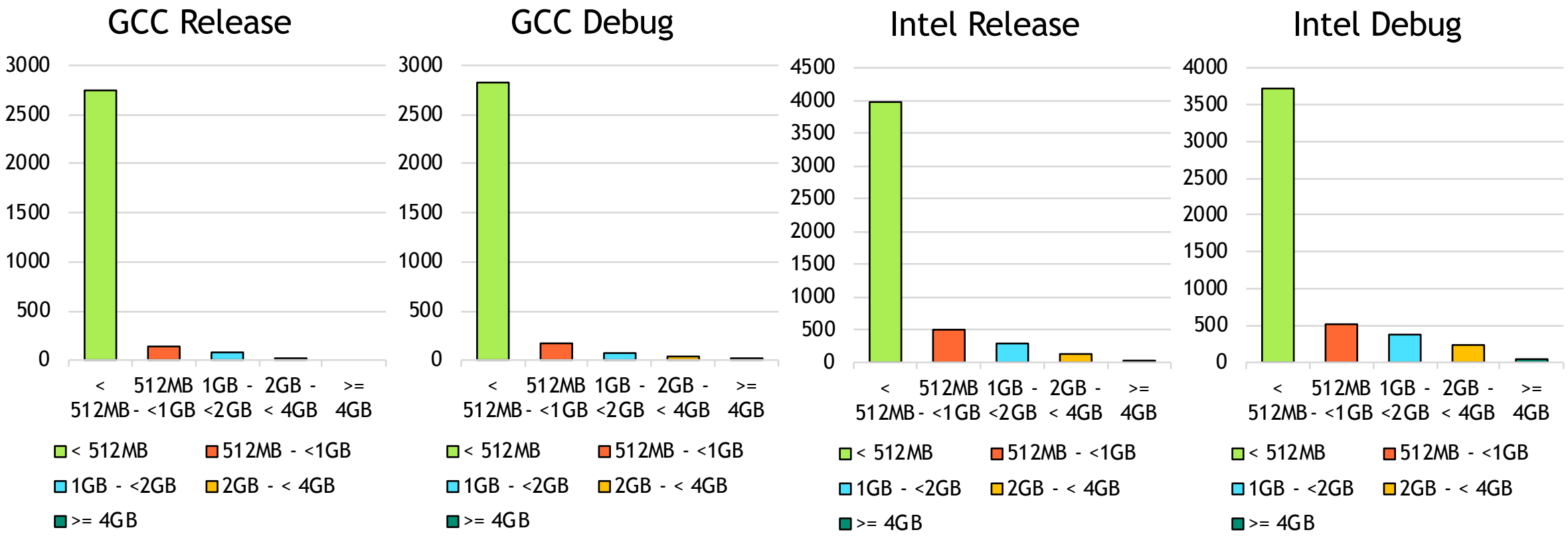
# Analysis – Design of User Service Nodes (Memory Required)




Typical user login nodes provide around 2GB – 4GB per core (compiler instance) on most systems but this needs to run O/S and other services too

**How much memory do large projects requirement for compilation on user service nodes?**

# Analysis – Vendor Compilers (Trilinos, Memory Required)




Typical user login nodes provide around 2GB – 4GB per core (compiler instance) on most systems but this needs to run O/S and other services too

**How much memory do large projects requirement for compilation on user service nodes?**

# Discussion: Compiler Memory Requirements

Larger variation in Intel compilers - correlates with user experiences that large parallel builds with vendor compilers can exhaust memory

- 128GB for user service nodes ( = 4GB per core max but need O/S and services)
- Can cause builds to crash if objects compile at the same time

Default on IBM XL is 8GB (for –O2, unlimited for –O3 and higher)

**As core counts continue to rise, NNSA procurements may need to consider larger memory user service nodes**

- Consider 2 – 4GB per core minimum
- Or will need to reduce build parallelism = reduced user productivity

# Object Sizes for Debug and Release

# Object Sizes by Compiler

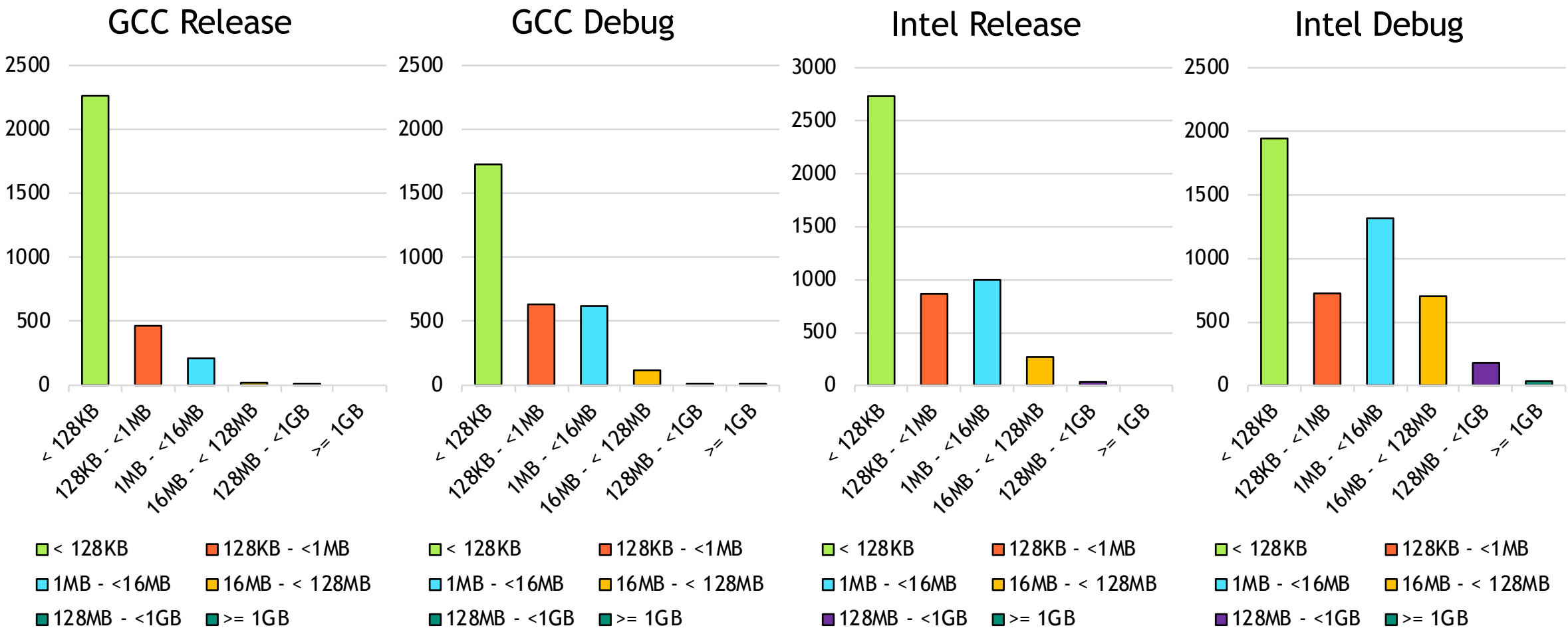
Anecdotal evidence is that object sizes have been slowly increasing for years

- More complex optimization leads to more code motion, more complex instruction generation (e.g. AVX512 has 15B instructions)
- Push within DOE for greater debuggability of problems has seen complex debug generation become standard
- C++ templating can create larger objects (can get multiple instantiations in some object files)
- Just more code … code bases increasing in complexity

**Problem:** some objects now so big this is causing issues with loading into debuggers, profilers and even during linking (libraries getting so large)

**How large are object files in a modern codebase?**

# Analysis – Object Size (Trilinos)




**How large are object files in a modern codebase?**

# Discussion: Object Sizes and Debug Symbols

Debug builds create much larger object sizes (inclusion of debug tables)

- These now exceed 1GB (for a single object file)
- Shows dramatic increase in size of debug information being generated
- Need debugging tools and profilers to be able to ingest very large debug tables
  - Historically this has been very challenging in CrayPAT, Intel VTune, Allinea MAP
  - NNSA needs to include support for large binaries in procurements for tools/systems

Evidence that significantly large object files create issues when generating libraries and linking

- Tables to find functions increasing, need to be loaded into memory

# Discussion

# Challenges for Moving Forward

**Developer productivity is critical to NNSA maximizing its investments and getting code bases ported to greater range of platforms in less time**

- Need to be more agile in our code development capabilities
- Recompile and Test needs to become faster
- Help with continuous integration methods (although even these are gated by long compile times and limited resource availability)

Traditional compilation methods are beginning to struggle in some areas

- Growth of complex C++ code base
- Templating and use of STL objects is causing lots more code to get generated, although not all of it gets used
- Debugging information is dilating build sizes (which then also take longer)

# Technology Options

DWARF5 debug format will allow external debug information to be generated (put into a separate file outside of the executable)

- Requires NNSA to push heavily on vendors to support new formats
- Clear change in how users expect to run tools (no longer a single package)

Other options that are appealing:

- Use JIT compilation methods to compile only what we need – shifts some of the compile time to runtime but dwarfed in traditional long running HPC simulations
  - **See Dave's presentation**
- Improve the quality of the compilers representations and mapping from front-end so we spend less time recreating programmer intent – more efficient code, faster compile times and potentially cleaner debug mapping
  - **See Pat's presentation**

# Tool Status

Build profiling tools are available at: https://github.com/sstsimulator/buildprofile

- Closed repo while tool development and cleanup takes place (please request access)
- Open once final development activities are completed

Development work on profiling tools will continue into FY19

Sandia plans to profile several (additional) large code bases in FY19 and may integrate the build profile into overnight reporting for some codes

Sandia Contacts: Si Hammond and Rob Hoekstra

- sdhammo@sandia.gov / rjhoeks@sandia.gov


Sandia
National
Laboratories